C++: New and Improved!

Last update: 2009-06-15

This talk summarizes the additions and changes forthcoming in C++0X, the next C++ standard. After briefly reviewing the C++ standards committee's approach, goals, and anticipated timeline for C++0X, we highlight several dozen language and library features, including concepts, concurrency, rvalue references, and uniform initialization syntax.

C++: New and Improved!

Walter E. Brown, Ph.D. `<wb@fnal.gov>`
Computing Division
Fermi National Accelerator Laboratory

2009-06-16

A little about me

- B.A. (math's); M.S., Ph.D. (computer science).
- Professional programmer for nearly 40 years.
- Experienced in both academia and industry:
  - Founded Comp.Sci. Dept.; served as Professor and Dept. Head; taught/mentored at all levels.
  - Managed/mentored programming staff for a computer reseller; self-employed as a software consultant and commercial trainer.
- At Fermilab since 1996; now in Computing Division/FPE Quadrant, specializing in C++ consulting and programming.
- Participant in the international C++ standardization process; Project Editor for forthcoming Standard on Mathematical Special Functions.
- Be forewarned: Based on the above training and experience, I hold some rather strong opinions about computer software and programming methodology — these opinions are not shared by all programmers, but they should be! ☺

Why should C++ change at all?

- "[W]e (the members of the committee) desire change because we hold the optimistic view that better language features and better libraries lead to better code."
  - "more maintainable"
  - "easier to read"
  - "catches more errors"
  - "faster"
  - "smaller"
  - "more portable", *etc.*
- "People's criteria differ, sometimes drastically."

— Bjarne Stroustrup, 2006

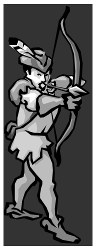
What do I mean by "new" and "improved" C++?

- Evolutionary advances via core language and standard library features that:
  - Extend or generalize C++03 behaviors, yet …
  - Interact well and compatibly with C++03 features.
- We designed these new and improved features to let us create programs that are:
  - Even closer to our problem domain (*i.e.*, at higher levels of abstraction), when that's what we want to do.
  - Even closer to our machines' architecture (*i.e.*, at lower levels of abstraction), when that's what we want to do.

WG21's approach to C++ evolution

- General principles:
  - Preserve source compatibility while improving performance.
  - Support novices, since $n_{\text{novices}} \gg n_{\text{experts}}$.
  - Help programmers to write better programs:
    - Maintain (and preferably increase) type safety.
    - Keep to the zero-overhead principle.
- Removing a feature is only rarely feasible:
  - Stability and compatibility are major concerns.
  - It's very hard to remove a feature without breaking somebody's code, but:
    - Keyword auto is so rarely used that we gave it new semantics.
    - Library's auto_ptr<> is heavily used but has inherent issues, so we deprecated it and provide/promote unique_ptr<> instead.

Goals



- For the core language:
  - Make C++ easier to teach and learn.
  - Make the rules more general and more uniform.
  - Make C++ better for building libraries; prefer libraries over language extensions.
- For the standard library:
  - Improve support for generic programming and other programming paradigms (styles).
  - Extend the library into new domains.
  - Apply the new core language technologies.

7

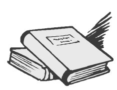
Features' status

- All have been formally adopted and balloted:
  - Detailed spec's are in the 2008-10 "Committee Draft" but …
  - Many details are still evolving, mostly in response to recent Ballot Comments by ISO members (National Bodies).
- Implementation experience:
  - Most features were based on existing practice in some compiler/library.
  - Quite a few were already in gcc 4.3.0 (released 2008-03).
  - But a few features are still not fully implemented anywhere, making some of us a bit nervous.
- Disclaimer:
  - Technically, anything could still change.

8

Planned timeline

- WG21 hopes to "resolve" (respond to) Ballot Comments by 2009-10:
  - Updated "Committee Draft" ⇨ "Final Committee Draft"
  - SC22 balloting by ISO members planned for early 2010, then:
    - Resolve any new FCD Ballot Comments, …
    - Updated "FCD" ⇨ "Final Draft International Standard" …
    - Final ballot at the JTC1 level.
- Hope to have the new C++ Standard out by very late 2010:
  - Could still encounter technical resistance.
  - Could still encounter political resistance.
  - Could still encounter publishing delays.



9

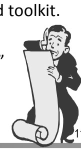
Also forthcoming …

- Some good library ideas/features have been delayed simply for lack of time to work on them:
  - Plan to issue these (*e.g.*, in Technical Report form) …
  - After C++0X is finalized.
- But final balloting is already under way for:
  - A *decimal arithmetic* TR (core language and standard library).
  - A separate International Standard for a *mathematical special functions* standard library:
    - First significant addition to <math.h> (and <cmath>) since ~1978.
    - Initially proposed by Fermilab for TR1 (2005).
    - I am the Project Editor for this Standard.
  - Versions of both are already adopted by the C committee.

10

Final disclaimers

- Today's survey of features emphasizes breadth over depth:
  - Not a tutorial; simplifies or suppresses many details, and …
  - Omits all background discussion (rationale, design issues, …). N2857
  - But identifies papers presenting such information →
    - At http://www.open-std.org/jtc1/sc22/wg21/docs/papers/
- Keep in mind that C++0X isn't designed to "fix" anything:
  - It's aimed at improving the C++ programming experience …
  - By improving/extending the programmer's standard toolkit.
  - "[T]he primary purpose of a programming language is to help the programmer in the practice of his art."
    — C. A. R. Hoare, 1973



11

New types and corresponding literals

- New long long integral types, signed and unsigned: N1811
  - sizeof(long long) ≥ sizeof(long).
  - Corresponding literals, *e.g.*, 42ULL.
- New pointer literal, nullptr, of type std::nullptr_t: N2431
  - A new name for the same null pointer value …
  - To avoid confusion with 0 as an int constant.
- New Unicode character and string types:
  - A char can now hold a UTF-8 character.
  - A char16_t, *e.g.*, u'x', holds a UTF-16 character; a char32_t, *e.g.*, U'x', holds a UTF-32 character. N2249
  - u8"Hello" is a UTF-8 string literal;
    u"Hello" is a UTF-16 string literal;
    U"Hello" is a UTF-32 string literal.

12

New integer type aliases (from C99)

- All found in new library header <cstdint>: N1835
  - Names all have the form int···_t or uint···_t .
  - Required iff some type has the ··· size/behavior.

Purpose	Examples	
Exact-width	int64_t	uint32_t
Minimum-width	int_least64_t	uint_least32_t
Fastest minimum-width	int_fast64_t	uint_fast32_t
Greatest-width	intmax_t	uintmax_t

More C99-compatibility features

- Preprocessor additions:
  - Variadic macros N1653
  - Empty macro arguments
  - Concatenation of narrow/wide strings
- Library additions:
  - Lots of additions to <cmath>, <cstdlib>, <cctype> N1568
  - In <complex>: acos, asin, atan, acosh, asinh, atanh, fabs
  - New <cfenv> to control the floating-point environment

Syntax to improve utility of existing C++ features

- Consecutive closing angle brackets now okay: N1757
  - typedef std::vector<std::vector<int>> Table;
- New for variant to iterate over a complete sequence:
  - int a[ ] = { ··· } ;
for (int & x : a) *// traverse entire array* a*, 1 element/iteration* N2394
x *= 2;
  - Works with any sequence that has explicit begin( ) and end( ) (*e.g.*, std::vectors) or implicit equivalent (*e.g.*, arrays).

New syntax for common function definitions

- Compilers can today generate default definitions for some c'tors, assignment op's, and d'tors:
  - But this happens only in the absence of your own def'ns.
  - Can now define these member functions via = default .
- A class is today made non-copyable by declaring the copy functions private, and not defining them:
  - Can now define these (and more) functions via = delete .
  - Calling a deleted function produces a compilation error.
  - class C { *// non-copyable*
C( ) = default; *// special mbrs only* N2346
C(C const &) = delete ; *// any function*
C & operator = (C const &) = delete ;
};

New flexibility in declaration syntax

- Type deduction from initializers via auto: N1984
  - std::vector<int>::iterator it = v.begin( );
can now be written auto it = v.begin( );
  - Uses same type deduction rules already used for templates.
- New permitted function declaration syntax: N2541
  - auto f (double) –> std::vector<double>; *// "auto" ⇨ [ ] ?*
- Type queries via decltype ("declared-type-of"): N2343
  - typedef decltype(x * y) result_t;
  - Especially useful in generic programming, when type interactions are often not known to the programmer:
    - template< class T, class U >
auto product (T t, U u) –> decltype(t * u) { return t * u; }

Feature completion: compile-time assertions

- Known as static_assert(···, "···"); N1720
  - Inspired by/augments run-time assert( ) macro and compile-time #error directive.
  - May appear at namespace, block, or class scope.
  - Evaluated strictly at compile-time, so has no run-time cost.
- Takes a predicate and a string literal; emits the literal as a diagnostic if the evaluated predicate is false:

```
template< typename T >
struct Check
{ ···
   static_assert( sizeof( int ) <= sizeof( T )
                , "Check: type is too small" );
};
```

Feature extension: template aliases

N2258

- Extends typedef notion.
- Adopts/extends *alias-declaration* syntax:
  - Syntax used today for only namespace aliases.
  - using *identifier* = *type-id* ;
- Now extended to templates:
  - ```
    template< class >
       struct A { ⋯ };
    template< class T >
       using B = A<T> ;   // B is now an alias for template A
    ```
  - ⋯ B<int> ⋯ *// now same as* A<int>

19

Enhancements to constructors

N1986

- A c'tor may now delegate to (make use of) another c'tor from the same class:
  - ```
    class C { ⋯
       C( ) : C( 3.14 ) { }
       C( double d ) ⋯
    };
    ```
  - Reduces/avoids duplication among c'tors within a class.
  - Eliminates need for special initialization member functions, since one of the c'tors can now serve this purpose.
- A derived class may now inherit its c'tors from a base class: N2540
  - ```
    class D : public B { ⋯
       using B::B;  // declare (inherit) B's non-default, non-copy c'tors
    };
    ```

20

Feature completion: new reference types

N2118

- T && is notation for new rvalue reference types:
  - C++03 *reference* types T & renamed *lvalue reference* types.
  - Allows code to distinguish between a memory cell (lvalue) and its contents (rvalue).
- Enables move semantics (*e.g.*, via std::move( )):
  - When copying is inappropriate or unnecessary or too expensive, can now instead <u>transfer</u> resource ownership.
  - ```
    class C { ⋯ // movable
       C( C && );                  // "move c'tor" overload
       C & operator = ( C && );    // "move assignment" overload
    };
    ```
- Also enables perfect forwarding (*e.g.*, via std::forward( )).

21

Generalization: compile-time constant expressions

N2235

- Means of declaring that an expression (not necessarily integral) be evaluated <u>by the compiler</u> whenever possible:
  - Can declare constexpr variables and (within limits) functions.
  - ```
    constexpr double sqr( double x ) { return x * x; }
    constexpr double gamma = sqr( 2.5 );
    ```
- Evaluated at compile time iff the argument can itself be evaluated at compile time:
  - ```
    double const alpha = 2.5;
    constexpr double gamma = sqr( alpha );  // okay
    ```
  - ```
    extern double beta;
    constexpr double delta1 = sqr( beta );   // error!
    double const delta2 = sqr( beta );       // okay; runtime
    ```

22

Enhancements to initialization

N2756

- Class member initializers:
  - Today limited to static data members of integral type.
  - Now extended to non-static and non-integral data members.
  - Avoids gratuitous inconsistencies between c'tors.
  - ```
    class C { ⋯
    private:
       double d = 3.14;
       C *    p = nullptr;
    };
    ```
- Uniform initialization syntax (see next 2 pages).

23

Generalization: uniform initialization syntax

- How can we initialize a variable of type T with a value v?
  - ```
    T t1 = v;        // copy-initialization (copy c'tor or equivalent)
    T t2( v );       // direct-initialization
    T t3 = { v };    // initialize from C-style initializer list
    T t4 = T( v );   // make a T out of v, then copy that T to t4
    ```
  - Today, different definitions of T allow 0, 1, 2, 3, or all 4 of these definitions to compile for identical v!
- Every C++0X initialization now accepts a { ⋯ } initializer:
  - To initialize free, base, member, or newed objects, as well as function parameters and return expressions.
  - Syntax also accommodates new type std::initializer_list< T > . N2531
  - Syntax also addresses some long-standing issues, such as:
    - The desire to initialize a std:vector< > with a sequence of values.
    - The "most vexing parse": T x ( ); *// does not default-initialize x!*

24
    ```

Uniform initialization syntax at work

- T v = { 1, 2, 3.14 } ; *// a free automatic variable*
  T * p = new T { 1, 2, 3.14 } ; *// a dynamically-created variable*
- void f1(T); f1({ 1, 2, 3.14 }); *// pass by-value*
  void f2(T const &); f2({ 1, 2, 3.14 }); *// pass by-const-lref*
  void f3(T &); f3({ 1, 2, 3.14 }); *// error: pass by-lref*
  void f4(T &&); f4({ 1, 2, 3.14 }); *// pass by-rref*
- T g( ) { return { 1, 2, 3.14 } ; } *// return by-value*
- class D : public T {
  T m;
  D( ) : T { 1, 2, 3.14 } , m { 1, 2, 3.14 } { } *// base, member*
  };

25

New language feature: concepts

- Inspired by standard library's *requirements tables*:
  - Concepts are notionally described as a type system for types.
  - Leads to vastly improved diagnostics when an algorithm is instantiated with a type not matching the algorithm's needs.
- The core language provides mechanisms to:
  - Articulate a set of requirements/constraints for a type, … N2773
  - Impose, *a priori*, such requirements on a template, and …
  - Define how a type meets such requirements.
- The standard library provides:
  - A library of standard concepts (mostly replacing today's extra-linguistic requirements tables), and … N2737…
  - Concept-based requirements for each standard algorithm.

26

Concepts at work

- *// Articulate requirements/constraints for a type* T*:*
  concept std::Swappable< typename T >
  { void swap(T &, T &); };
- *// Impose requirements on the type used to instantiate this algorithm;*
  // implement the algorithm using only those requirements:
  template< typename T >
  void fancy_sort(T * from, T * upto)
  requires std::Swappable<T>
  { ⋯ ; swap(*p, *q); ⋯ ; }
- *// Define how* int *will meet the* Swappable *requirements:*
  concept_map std::Swappable< int >
  { void swap(int & t1, int & t2) { std::swap(t1, t2); } }
- *// Instantiate the algorithm on* int*; okay since* int *is* Swappable*:*
  fancy_sort< int >(a + 0, a + n);

27

New feature: concurrency

- Intended to standardize support for:
  - Multi-core processors.
  - Client-server programming.
  - Current POSIX and Windows standards re OS threads and shared memory …
  - But not replace other standards (*e.g.*, MPL, OpenMP, …).
- The core language now answers such questions as:
  - What does it mean to have two threads sharing memory?
  - How does this affect variables?
- The standard library now answers such questions as:
  - How are threads created/synchronized/terminated?
  - How are exceptions handled between threads?

28

Concurrency in the core language

- Uses "loosely-coupled shared memory" as a model: N2480
  - Today's hardware does not support stronger coupling.
  - A data race (*e.g.*, multiple threads updating a single object) will evoke undefined behavior if not protected.
  - Selected atomic (indivisible) operations are provided and also have a library-style interface.
- New thread lifetime storage duration:
  - New keyword thread_local indicates that a static variable is to go away when its thread ends. N2659
  - A variable declared thread_local static has a single instance per thread (whether at namespace, block, or class scope).

29

Concurrency in the standard library

- Thread instantiation via basic std::thread type:
  - Supports creation and join( ) operations. N2497
  - Supports standard access to OS-specific details.
  - Supports thread synchronization:
    - ① Via variables of type mutex, as well as …
    - ② Via condition variables.
- Thread termination is voluntary:
  - Synchronous in all cases, with no interrupts permitted.
  - Typical thread termination is via return from the function called when the thread was initiated.

30

Saying hello

```
#include <iostream>

void greet( ) {
   std::cout << "Hello, world!\n"
}
```

```
// In a serial world:

int main( ) {
   greet( );
   return 0;
}
```

```
// In a parallel world:

#include <thread>

int main( ) {
   std::thread t { greet };
   t.join( );
   return 0;
}
```

31

And there's still more!

- Variadic template parameters: N2080
  - Now possible to define a template taking a variable number of template parameters (a "parameter pack").
  - Useful to implement library facilities, *e.g.*, std::tuple< >.
- Strongly-typed enums (also termed scoped enums): N2347
  - enum class E : long { E1, E2, E3 = 100, E4 /* = 101 */ };
  - E1, *et al.*, are exclusively in E's scope (*i.e.*, E::E1).
  - Enumerators have an underlying integral representation, whose type the programmer may optionally specify.
- All template parameters may now have defaults: Core#226
  - template< class T = long double > *// not allowed in C++03*
    T pi ( ) { return 3.14159265358979323846264 33L; }

32

Monomorphic lambda expressions and closures

- Anonymous function objects (lambdas): N2550
  - Definable at point of use.
  - May be local to (nested within) another scope, and can capture (use) local variables by value or by reference.
  - Feature loosely based on Alonzo Church's λ-calculus [1936].
- Examples:
  - ```
    std::transform( v.begin( ), v.end( ), v.begin( )
                  , [ ] ( double x ) { return x + pi( ); } );
    ```
  - ```
    auto add_pi_to = [ ] ( double x ) { return x + pi( ); } ;
    std::transform( v.begin( ), v.end( ), v.begin( )
                  , add_pi_to );
    ```

33

And yet more language additions, improvements, …

- Relaxation of POD restrictions; new notion of trivial type.
- New alignof (⋯) data alignment support.
- Raw string literals.
- Extension to obtain the size of a data member via sizeof.
- Extensible (user-defined) literals.
- Generalized attribute declarations.
- Improvements to union ("Toward a More Perfect Union").
- Conversion operators may now be declared explicit.
- Extended friend declarations.
- C99 preprocessor semantics.

34

And I've barely mentioned the standard library

- Random number engines and distributions (by Fermilab!).
- Regular expressions.
- Type traits (for template metaprogramming).
- Posix-related enhancements to standard exceptions.
- Generic callable wrapper function< > and binder bind< >.
- Smart pointers: shared_ptr< >, unique_ptr< >.
- Containers: array< >, tuple< >, and forward_list< >.
- Hash tables: unordered_map< >, unordered_set< >, *etc.*
- Concurrency support: atomic< >s, threads, mutexes, *etc.*
- New variadic min( ), max( ), and new minmax( ) algorithms.

35

Summary

- Lots and lots of useful improvements are coming in C++0X:
  - Missing a few hoped-for items (*e.g.*, garbage collection), …
  - But WG21 already has a heavy workload.
- Compilers (*e.g.*, gcc) and libraries (*e.g.*, Boost) already have very many of the new features available:
  - We can start now to learn/try out the new features, and …
  - We can start now our planning for a transition.
- "C++ is a general purpose programming language for enjoyable programming by serious programmers."
  — Bjarne Stroustrup, 1991

36

C++ : New and Improved!

FIN

Walter E. Brown, Ph.D. <wb@fnal>
Computing Division
Fermi National Accelerator Laboratory

2009-06-16